AN EXECUTION MODEL FOR EXCEPTION HANDLING IN A MULTI-AGENT SYSTEM

Zina Houhamdi[(a)], Belkacem Athamena[(b)]

[(a)] Software Engineering Department, Al Ain University of Science and Technology
Al Ain Campus, UAE
[(b)] Management and MIS Department, Al Ain University of Science and Technology
Al Ain Campus, UAE

[(a)]z_houhamdi@yahoo.fr
[(b)]athamena@gmail.com

ABSTRACT
Multi-Agent Systems (MASs) are required to exhibit diverse quality attributes like robustness, flexibility, and possibly the ability to accommodate to their agents and context dynamics without external intervention. Exception supervision contributes to the achievement of these goals, and the agent society has proposed many approaches and patterns to supply MASs with exception handling skills. This paper is dedicated to studying this specific subject, particularly in knowledge-based agents systems. The research aim is bilateral: the first aim is to understand the exception concept in MAS. Therefore, we can determine the study objectives to discuss. The second aim is to examine part of these objectives. The suggested methods in this paper describe approaches and outcomes which are estimated to support the agent society and ultimately to support Software Engineering, possibly included within an undergo evolution manner. Previous investigations have concentrated primarily on the systematic perspective of exception handling. Our methods propose to introduce exception utilities in the MAS context.

Keywords: multi-agent system, exception, exception handling

1. INTRODUCTION

MASs consist of multiple autonomous entities called agents, each having different information and/or diverging interests. They are distributed, and complex systems and the agent society try to achieve collaboration and competition between agents to perform their actions in an extremely easy and modular manner. The agent technologies are very widely applied, and we can find in the literature several applications varying from software agents that support people across the network to independent robots in industry. Consequently, MASs are an optimistic approach and technological advancement in artificial intelligence and software engineering (Chopinaud et al., 2006; Houhamdi, 2011).
Since MASs are considered principally as software, and according to computer science history of the past fifty years, the development of reliable systems needs devoted effort, attempts and exercises. Reliability is a system quality measuring system convenience to the user, system accuracy to support the user requirements, and system execution performance and efficiency. Methods for fault detection were proposed and implemented in conventional software engineering to improve the reliability level of software. Contemporary accomplishments assure some of the previously mentioned characteristics in diverse circumstances of non-open and uniform systems. MASs defy existing implementations and focus on complicated applications because they are requested by the users of the software and the organization structure. MASs apply to systems which are heterogeneous, interactive and composed of independent agents.
Among methods to enhance the software reliability, exception handling is reputable and well known as a robust and simple technique (Castelfranchi, 2005; Houhamdi and Athamena, 2011a, 2011b, 2012). Exception handling was included in Programming Languages (PL) since a long time ago to manage unusual situations faced during the code running adequately and methodically. On the other hand, distributed systems have demonstrated that exception handling techniques need particular expansions to apply to these kinds of systems. Simultaneously, achievements in software development have increased the necessity for alternate methods also. MASs also possess characteristics that require re-examining the exception subject.
The purpose of this paper is to understand the concept of exceptions in MASs and to suggest an appropriate architecture for MASs which is open, heterogeneous and features mainly autonomous agents. The agent society has prompted many studies that demonstrated the necessity to handle exceptions in MASs at the system level. This handling includes management and necessary techniques encompassing the management. Solutions proposed up until now apply to a restricted set of MASs only, where usually agents are non-autonomous, and the methods at system level necessitate a perfect collaboration between agents

during exception handling execution (Chopinaud et al., 2006; Houhamdi and Athamena, 2011a). Agent autonomy is an essential quality that must be guaranteed when agents treat exceptions by themselves in the first place; this is considered a requirement of any proposed solution. In this case, exception handling then depends on mechanisms at agent level to manage the weakness of existing solutions and improve them. In an exception situation, the decision is taken by the agent itself to start treating the exception, trust in own expertise or request help from the system level.
The model proposed in this paper guarantees the agents' autonomy by ensuring that the agent maintains control during its processing even when exceptions occur. This approach allows the agent to make an individual decision if a situation is an exception; hence the autonomy is enforced. The approach is specified formally, and its relative architecture is described.

2. BACKGROUND

The actual solutions for exception handling in MASs give the illusion that agents can deal with a situation which isn't Exceptions at Programming level (PE), although they still require taking into account PE situations like unforeseen or uncommon states (Issarny, 2001; Romanovsky, 2001). This illusion is conducive to the notion of Agent Exception (AE) that is described in this paper. To investigate and define an appropriate signification of AE we use the primary definition of PE as starting point. According to function oriented and object oriented PLs, the word "exception" has obtained a specific definition, firmly joined to programming standards, exemplified by the Good enough definition (Goodenough, 1975):
Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation's invoker. The invoker is then permitted (or required) to respond to the condition.
Whenever a program makes a procedure call during its runtime, the procedure needs to evaluate conditions that must be valid before execution. If one condition in the minimum is not approved, the procedure sends a note to the caller declaring that it cannot be performed because of the condition infraction.
Since the MAS constituents (agents, resources, and context) are all programs, we can apply this definition. Nevertheless, the MAS properties and past studies prove that this definition is inappropriate to deal with AEs, due to the autonomy, heterogeneity and openness features. The previous definition of exception constrains the called procedure to assert definitively that a circumstance is unusual. This approach is inadequate to MASs, where ambiguous understanding is likely to arise. An agent is assumed to be autonomous when it can make a decision alone. The interpretation of PE doesn't grant this decision, as represented in Figure 1.
If the method decides that there is an exception, the caller doesn't have an alternative choice to execute. For instance, the method response is an exception object in several object-oriented PLs. However, this solution does not outline the purpose for AEs, as illustrated below in Figure 2.

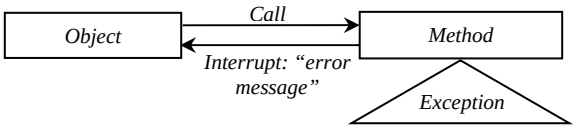

Figure 1: Programming Exceptions

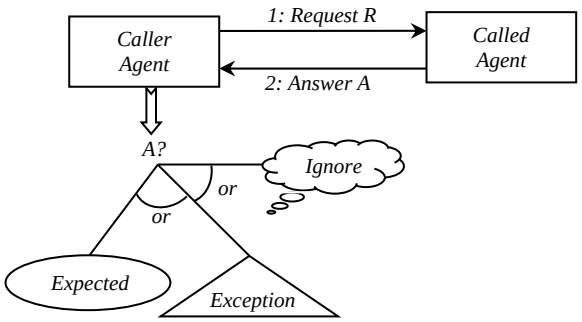

Figure 2: Agent Exceptions

Independent agents must be qualified to determine if a received message from other agents (initially or following a demand) is usual, unusual, or, for example, to be disregarded. Therefore, this declaration is extended to every received message by an agent from agents, the context, or the external environment components.

- *Agent Exception*: By MAS traits, the AE model described in this paper is presented and understood at the agents' level. In other words, the basic element in the exception handling is a whole agent unit, not just the instructions set in its program.
- *Definition*: An AE is the agent's understanding of an observed situation as unusual or not expected.

The previous definition describes the agent's influence in the exceptions situation and, more precisely, during the decision step directly related to the events observed by the agent (Figure 2). During the reception of an event, the agent can determine how to arrange this event. This belief is the main criterion for exception decisions. The agent is a knowledge based entity that performs a protocol. The agent goals and tasks permit the expression of expectations for context modification in the future: agents communicate by sending messages for the purpose of obtaining special outputs that are considered as expectations. Accordingly, the agent is qualified to classify a received message as unexpected if the message does not agree with its expectations.
The PEs are different from AEs. The former concerns the event and the later concerns the event understanding. Agents, which are autonomous, can

control themselves and determine how to deal with events; this is the principle of AE.
Someone can be in dispute with this definition and argue: independent agents are usually assumed to operate in a society setting. A society establishes strong relations. These relations assume that, even with independency, agents act based on received demands. Such context is relevant in a closed systems environment in which a human user supervises all system components. The illustrative supervisor-worker pattern assumes that workers comply with the supervisor. However, in an open-system, the designer of particular agents desires to preserve total control of its agents and wants to decide on how to reply to requests from other, possibly anonymous, agents. Notwithstanding strong relations among two agents, independency is conducive to the previous definition. It is the responsibility of the agent alone to determine how to deal with an event.
This definition does not go against the strong relations decided by societies. Coordination ability is merely considered as an extension of agents' autonomy. When an agent has concluded that the event is unusual to its understanding, the agent can improve its conclusion based on strong relations. For instance, an 'operator agent' can decline to abort when the mandate is from a 'supervisor agent,' for example, if the two agents work in separate organizations but share a virtual space for cooperation.
In MASs, AEs are associated with agent tasks, and they influence the agent level. PEs are related to situations. They influence the agent at the code level. This description is illustrated in Figure 3.

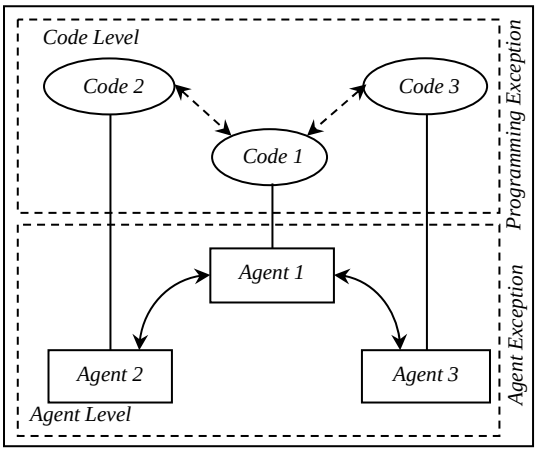

Figure 3: Exception Levels in MAS

The following section's purpose is to determine the exception area of agents and to describe the relationships between the PE and AE.
Note that the PEs can cause AEs. By way of illustration, an unforeseen agent ending as result of a PE, such as a null pointer, impacts the system organization directly. This PE will then cause an AE, 'agent death' (Houhamdi and Athamena, 2011a; Klein et al., 2003). In this situation, the remaining agents require rearranging their tasks to compensate for the agent death. In this manner, the rearrangement is an exception that happens at the agent level.
However, some PEs, occurring in an agent, will not generate an AE. As case in point, network exceptions, where a handler retries the network connection to approach this issue, are commonly controlled at the code level. Accordingly, the agent pursues performing its task.
Nevertheless, AEs do not generate PEs. Particularly, agents are not aborted by the AE's occurrence. That is to say, AEs do not provoke the agent code to contend with a malfunction. AEs do not cause PEs because AEs are discovered in input messages using a particular estimation method. The message is treated as an AE, while the program is correctly performed and no PE is revealed. The agent proceeds its activity by managing the anomaly or disregarding the message and continuing the following iteration. During this method, the agent status and its code are consistent with the typical progression without producing any PE.
The previous characteristics single out one-sided relations among the two exceptions kinds, which are illustrated in Figure 4. It shows the relations between the exception areas that are conceived for a MAS. The PEs can, in some situations, generate an AE, although the opposite is impossible.

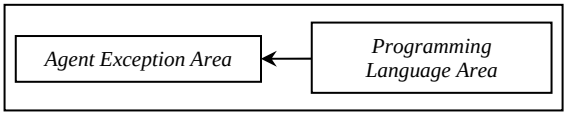

Figure 4: Exception Area Mapping

MAS Exception: The relation between PEs and AEs allows classifying the uncommon circumstances that confront an agent. This section purpose is to identify exception classes to make their investigation easier and also to organize the handlers' classes that will be produced.

- *The Identification Axis*: Mainly, there are two classes of exceptions depending on if the exception is identified or unidentified. If there is a handler to manage the exception then it is classified as identified; otherwise, the exception is unidentified. In PE, unidentified exceptions provoke an early program ending, because it cannot manage the event and possibly risks damage to the physical devices or operating system. However, in AEs, unidentified exceptions signify that the agent lacks the necessary knowledge to manage the event in the present situation. Nevertheless, the agent state is still reliable and can make a decision by its qualifications. The simplest solution is to not pay attention to the event (like Does-Not-Understand in Smalltalk), but the complex solution is to exploit the circumstance, like KGP agent (Antonis et al., 2004).

- *The Coverage Axis*: According to coverage of the agent exception, we define two types: Alone and Team. If the agent can manage the exception without the assistance of other agents, the level is labeled as Alone. If the agent needs to interact with other agents to manage the exception, the level is thus assumed as Team. In an agreement contract, if a customer gets a remarkable bid, for example, inferior 15% of the estimated cost proposed by the customer, in this case, the exception level is Alone and can be managed quickly. The customer updates its status so that this bid will gain the call for proposal. At this point, the customer continues the procedure execution to agree on the bid officially and reject the remainders without additional cooperation needed to manage this special circumstance, so this exception is an example of the Alone type. On the other hand, an exception such as declaration postponement is an example of a Team exception. A supplier declares a deferment to the customer, who replies by allowing a deadline prolongation to every supplier.
- *Handler Description*: In this section, we define the exception classes, and then we categorize the applicable handlers. By default, the death of an agent is presumed as a common situation that is classified as an identified AE (Klein et al., 2003; Miller and Tripathi, 2004). Still, this AE can be dealt with in either an alone or team manner, according to the handler type used by the agent to control the situation.

There are two objectives behind exception organization. The first one is to guide developers to design handlers or methods to elaborate during execution. Based on the MAS application, certain handlers' types are essential and others are unessential. Handlers for the identified exception need particular methods to seek or create them; this is very expensive for some systems. The other objective is to help agents in the decision process. Based on the exception, the agent examines a special handler.

Handlers' classes are described by the abbreviation in Table 1. For example, IAA refers to handlers for identified exceptions at Agent Alone level, while UC represents handlers for an unidentified exception at Code level.

Table 1: Exception Classes

	Agent level		**C**ode level
	Alone	**T**eam	
Identified	IAA	IAT	IC
Unidentified	UAA	UAT	UC

- *Handlers Classification*: Since there are two manners to manage exceptions (Alone or Team), agents will confront a difficulty in deciding the availability of the handler of every category. Team handlers are a costly process, particularly in distributed systems, and they overcome the distribution advantages because they extend the computation cost with interaction expenses. Therefore, the agent prefers handlers that handle exceptions in an Alone style. Further, the interaction complexity in MASs accentuates this choice.

The exception handling method emphasizes representing exceptions on the communication protocol because "MAS" principally refers to cooperative agents that perform as reported by the communication protocol. In this paper, we use the word "exception" to denote AE if there is no confusion with PE.

3. AGENT EXECUTION MODEL

The AE definitions have impacted the agents' execution model and framework. The best agent frameworks use the Belief-Desire-Intention model, including the Jason and Jadex architectures or the KGP architecture. However, these models suffer from two weaknesses regarding AEs. Exception handling is not processed explicitly in the agent execution model and no distinction is made between exceptions. The exception is treated as PE and depends upon the languages services. The AE's handling, on the other hand, needs to consider the MAS properties, and the best practices propose to distinguish clearly between the application logic and the exception handling. This work proposes an agent execution model which includes exception handling so that the previous distinction is achieved.

In general, the execution model for MAS is iterative, traditionally a cycle of perception, reasoning, and action. Our model uses the same iteration but extends the perception and action activities to relevantly arrange the reasoning activity in exception situations, considering the agent independency property.

The remaining of this paper defines the proposed agent's framework mainly by describing its protocols, handlers, and knowledge, and after that the execution model.

Protocol and Handler Structure: AUML and allied studies have modeled handlers and protocols using sequence diagrams or graphs. We decided to describe handlers and protocols as diagrams (more specifically directed trees) to set up the description formally. The root represents the initially transmitted message. The tree is organized by using the relation *R,* defined as follows: If T is a directed tree; L represents the leaves kit $(L \subset T)$ and M the edges kit. The edges represent operations such as sending a message in handlers and protocols.

R is a non-symmetric, non-reflexive and transitive binary relationship. T verifies the following structural properties:

1) $\forall m \in M \setminus L, \exists m' \in M, m\ R\ m'$
2) $(\forall m \in M \setminus L, suc_T(m) = \{m' \setminus m\ R\ m'\}$

ISBN 978-88-97999-91-1; Bruzzone, De Felice, Frydman, Longo, Massei and Solis Eds.

3) $\forall m \in M \setminus \{root\}, \exists m' \in M, m' \, R \, m$

The first definition declares that all sent messages have a successor except leaves. $suc_T(m)$ represents the successors set for a given edge of T in definition two. Definition three states that all sent messages have a predecessor, except the root.

In the case where protocol comprehends a loop in its description, the tree specification utilizes the cycles unrolling over the tree branches. Such unrolling action is usual, e.g. Petri nets.

We describe two sets: the M messages Set, the H histories Set, and $\phi \in H$ (Empty execution). The execution continues based on the acquired message kind and the handler (h) and protocol (p) state which the agent executes. "*Perform*" defines the progress of the agent running the protocol and the handler.

Perform: $M \times H \times H \rightarrow H \times H$

$$(m, H_p, \phi) \quad \begin{cases} (H_p \cup \{m\}, \phi), if \ m \neq end \\ (\phi, \phi), if \ m = end \end{cases}$$

$$(m, H_p, H_h) \quad \begin{cases} (H_p, H_h \cup \{m\}), if \ m \notin \{end_p, end_h\} \\ (H_p, \phi), if \ m = end_h \\ (\phi, H_h \cup \{m\}), if \ m = end_p \end{cases}$$

(m, H_p, ϕ) describes the protocol p's execution. The execution history evolves during message processing (sent and received), and the processing terminates when the end is obtained; in this case, the protocol history is cleaned out but (m, H_p, H_h) represents a handler execution. Consequently, the handler treatment follows the protocol execution. When m is end_p, the handler starts after the protocol interruption. Finally, when m is end_h, the handler processing is completed with success, and the protocol execution is restarted.

Figure 5 represents a general execution model of agents which contains three layers. We explain them consecutively in the following paragraphs. The description depends on algorithms which are independent of the application domain.

First Layer: This layer contains message reception, pertinence checking and belief comparison, which are the basic phases in the agent processing model. The agent collects the messages from its inbox. They are sent to pertinence checking to discard messages that are not important for the agent, as reported by the relevance table. Pertinent messages are matched to the agent beliefs in the *Beliefs Table*. If an equal entry is located, then the output is an expected message, otherwise *Taking Decision* is started when the message is unexpected, and the *Handler Selection* is activated.

In the *Decision Process*, which is the agent's brain, the message is treated to define the agent action, if any, and update the agent knowledge. Besides this task, the *Decision Process* performs continually and does not need an input to generate an output. This function is not illustrated in Figure 5 because it is not related to exception management. Nevertheless, it is essential because it is the 'dynamic' part, indispensable for the agent to induce actions.

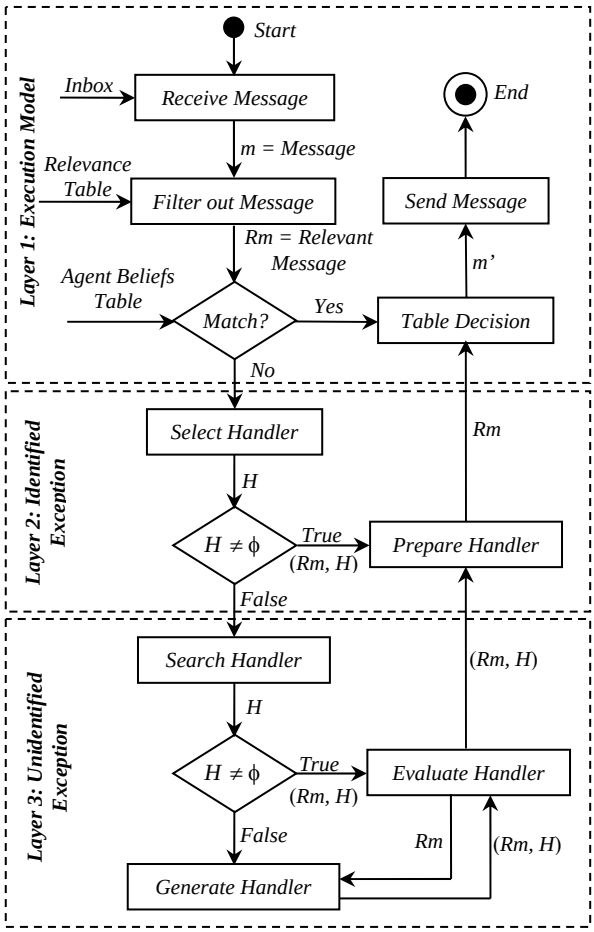

Figure 5: Agent Execution Model

Second Layer: The agent finds an unusual situation when a match is not found in *belief matching* phase:

- *Handler Selection*: concerns *Identified Exceptions*, i.e. the agent possesses a handler to manage the found exception. Unforeseen messages are forwarded to the *Handler Selection*; this later explores the handler table for a relevant handler. If a table entry has a requirement that meets the message, a handler is located. If diverse handlers are located, the *Favorite* method determines which handler is better for the agent, according to its environment and state. The *Favorite* method is thus domain dependent. *Favorite* methods use metrics to appraise handlers (e.g., the handler complexity).
- *Handling Preparation*: when a handler is located, it is sent to the *Handling Preparation* phase which interrupts the protocol affected by the unexpected message, starts the execution of

the handler and specifies that the interrupted protocol needs to be appraised at the termination of the handler by producing a protocol dominion for the handler. Thus, the agent decides whether to continue the interrupted protocol or to abort it. The output of this phase is a message which is sent to the *Decision process*, apt to treat the exception.

Third Layer: If the selection phase does not find a handler, the agent faces an *Unidentified Exception*; in other words, the agent does not possess a handler:

- *Handler Search & Evaluation*: The agent attempts to find a *Handler* by communicating with other agents or with a handler depository. A successful search provides a handler. The *Handler Evaluation* process analyses the handler fitness, keeps the agent autonomy concerning this foreign handler, and saves the exception class and its handler in the handler table for future use. Usually, the evaluation process is complex but we use a simple method by assuming that a handler is considered adequate if it reaches a situation allowing the suspended protocol to continue its processing.

Formally, the handler adequacy, *H*, is adequate if and only if $H_n = P_{it}$,

where *H* is a Handler: $H = (H_i), i \le n$

and *P* is a Protocol: $P = (P_i), i \le n$ interrupted at the statement P_{it}

and $end_p = end$

More precisely, the agent accepts the handler if it directs the processing to the desired situation before the exception occurrence. However, this basic verification does not certify that each statement in the handler is adequate for the agent. This generic approach is domain dependent.

- *Handler Generation*: In case the handler search fails to find a handler (or the found handler is unacceptable), the agent tries to generate a *Handler*. In the proposed approach, this phase unavoidably engenders a default handler H_d if no acceptable one is found; this H_d is important since it ensures the execution progression. The H_d will disregard messages during a period before declaring the failure of the protocol. For example, the handler H_g produced to expect the message m during two times related to the protocol P is described as:

$$H_g = \big((m, end_h) \big| m, \tau(ignore)\big), \big((m, end_h) \big| m, \tau(ignore)\big), \big((m \big| (m, \tau(update(p)), end_p, end_h\big)$$

The agent assumes it will get *m* two times, and then sends end_h to the protocol. Each time, the message is ignored by the agent if it does not match its beliefs. After three non-matching messages, the protocol state is updated and a message end_h indicating the protocol annulment is forwarded to all the agents.

Table 2 presents the model complexity in all cases, with the following legend: d_p (decision process), h_s (handler selection), *eval* (handler evaluation) and *p* (protocol).

Table 2: Model Complexity Table

Case	Complexity
No Exception Handling	n_{dp}
With Exception Handling	$n_{dp} = \max(n_{dp}, O(n_p))$
Identified Exception	$\max(O(n_{hs}), O(n_d))$
Unidentified Exception	$\max(n_{eval}, O(n_{hs}), O(n_d))$
Unidentified Exception Default Handler	$\max(n_{eval}, n_{dh}, O(n_{hs}), O(n_d))$

Since agents execute a few protocols concurrently, the cost is reasonable in comparison with MASs without exception handling. However, in the case of heavy agents, we should consider other approaches for fault tolerance.

4. AGENT ARCHITECTURE

Figure 6 represents the agent framework. It is comparable to existing agent architectures and it integrates special components for exception handling. In particular, these components can be taken out from the architecture if the agent does not need this feature or as a result of design choices.

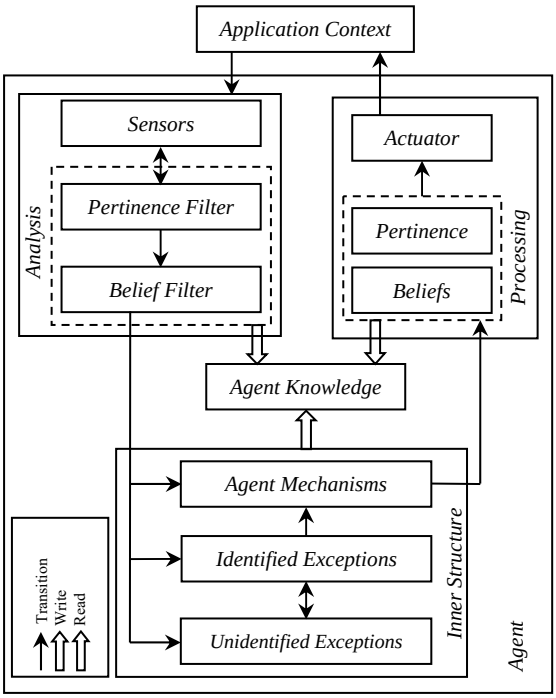

Figure 6: Agent Architecture

ISBN 978-88-97999-91-1; Bruzzone, De Felice, Frydman, Longo, Massei and Solis Eds.

The agent framework includes four principal components to be harmonious with the execution model: the internal description, internal processes, perception, and operations.
The correspondence between the execution model and the framework is illustrated in Table 3. The left column names the execution model components defined in the third section. The middle column lists the framework components described in the fourth section. Parts of these components are common architecture components and the others are arranged in different framework components in the right column.
Table 3 shows that the proposed agent architecture includes the required features to implement the whole execution model. The constituents of the architecture support two characteristics of agent models with exception handling abilities:

- The Architecture Components supply the developer with an advanced and general architecture prototype (Athamena and Houhamdi, 2017). The column guides architecture refinement.
- The Architecture Constituents separate the application concerns from the exception concerns.

Table 3: Correspondence between Agent Architecture and Agent Execution Model

Execution Model Constituents	Architecture Constituents	Architecture Components
Receive Message	Sensor	Perception
Filter out Message	Relevance Filter	Perception
Compare with beliefs	Expectation Filter	Perception
Take Decision	Base Mechanism	Internal Processes
Select Handler	Identified Exception	Internal Processes
Prepare Handler	Identified Exception	Internal Processes
Search Handler	Unidentified Exception	Internal Processes
Evaluate Handler	Unidentified Exception	Internal Processes
Generate Handler	Unidentified Exception	Internal Processes
Update State	Generation	Operations
Send Message	Operator	Operations
Tabular Knowledge	Internal Description	Internal Description

These two characteristics are essential for the developer since the refinement and separation of concerns are well-known as good practices in Software Engineering.

5. SIMILAR WORKS

Exception handling research covers investigations in artificial intelligence and software engineering. As MASs are also related to these areas, several tangible implementations are noticed in the MAS theory or their building practices. Nevertheless, these achievements do not satisfy the essential qualifications to approach MAS exceptions. PEs have established theories, but they are not applicable to MASs adequately because of MAS features such as heterogeneity, openness, and autonomy. These approaches can deal with the openness and heterogeneity issues; however, they cannot manage the autonomy characteristic. One remarkable effect is that there is no attempt to provide a precise description of the exception notion in a MAS, particularly in the agent society. Several illustrations are clarified in depth, e.g. agent death, but the exception notion remains implicit. The most notable works that approach exception handling in MASs are:

- *The Sentinel Architecture*: sentinels are agents inserted in a MAS software to supply the application with a fault tolerance capability level (Athamena and Houhamdi, 2017; Hägg, 1997). The Sentinel supports the agents in their communication. Sentinels are specially designed for fault detection and recovery. The detection of an exception during agents' communication activates the sentinels which try to resume a reliable situation. The problem is that the sentinel violates the agent paradigm assumptions (encapsulation is not respected and, consequently, neither is agent autonomy).
- *Sentinel-Like Agents*: extends the sentinel model with a reliability database (Klein et al., 2003) where the failed agents are stored. The database leads sentinels in recovery functions to reduce the needed time to recover. The Sentinels operate similarly to the Hagg initial model without inspecting agent interiors, as to enhance the agents' autonomy. However, this system suffers from two weaknesses: the agent autonomy is violated (as Sentinels can change agent message), and the exception handling system is fragile in the case where sentinels cannot perform their activities when executing a handler.
- *Commitment Protocols*: consider exception management in the business milieu (Mallya and Singh, 2005). This model uses commitment protocols to describe the agents' communication in an open system. This approach preserves the agents' autonomy. However, it is principally abstract, and it requires validation in real world applications.

SaGE in the Mad-Kit Platform: SaGE is a framework which adds to the Java exception management system services to manage problems related to autonomous agents in the Mad-Kit (Souchon et al., 2004). In Mad-Kit, an agent possesses roles and provides services to other agents. Exceptions can happen at the role, service and agent level.
SaGE follows the agent exception description, but does not ascend to the heterogeneous system level because it uses just benevolent agents. However, SaGE contributes notably to agent-oriented engineering by including exception handling, to wit the exception expansion according to the particular organization model and the cooperative exceptions.
Our approach endows an individual agent with relevant potentialities concerning exception situations and conforming to agent characteristics. Existing systems satisfy part of the agent features, but our model approaches the autonomy issue appropriately. The principal model advantage compared to other systems is

its robustness and reduction of the developer job; in this manner, the developer will be able to focus on more important processing matters.

6. CONCLUSION

MASs should have many features such as robustness, flexibility and automatic adaptation to the agents' dynamics and context. Exception handling is one of the techniques that contribute to the achievement of these features, and the agent society has proposed diverse approaches to supply MASs with exception handling facilities.

Agent exceptions need certain special approaches to assist developers in writing pertinent handling programs. Our model improves the agent framework's ability to analyze the messages and identify the unforeseen ones during collaboration protocol. The presented approach is integrated into the agent framework to allow the developer to concentrate on writing suitable handlers' programs. The model uses these handlers to manage exceptions whenever needed. The framework supplies agents with the model, thus they treat exceptions autonomously.

The proposed approach discusses exception handling at the agent level, which treats agent level and system level exceptions in a decentralized manner (complex and inefficient), however strong and adaptable if the MAS faces exceptions. The agent level hides the problems related to the system robustness because agents are autonomous and the MASs are open and heterogeneous, and the system level improves the system performance.

Finally, a future improvement of the proposed approach is an extra investigation into the handler generation methodologies in different circumstances to make the agents more autonomous when encountering diverse, unusual events. Further, an interesting domain is the agent evaluation of management approaches proposed by the collaborative agents in the MAS.

REFERENCES

Antonis, C., Paolo, M., Fariba, S., Kostas, S., & Francesca, T. (2004). The KGP model of agency. In the 16th European Conference on Artificial Intelligence (pp. 28–32). IOS Press.

Athamena, B., & Houhamdi, Z. (2017). An Exception Management Model in Multi-Agents Systems. Journal of Computer Science, 13(5), 140–152.

Castelfranchi, C. (2005). Mind as an Anticipatory Device: For a Theory of Expectations. In International Symposium on Brain, Vision, and Artificial Intelligence (pp. 258–276). Springer Berlin Heidelberg.

Chopinaud, C., El Fallah-Seghrouchni, A., & Taillibert, P. (2006). Prevention of harmful behaviors within cognitive and autonomous agents. Frontiers in Artificial Intelligence and Applications, 141, 205–209.

Goodenough, J. B. (1975). Exception handling design issues. ACM SIGPLAN Notices, 10(7), 41.

Hägg, S. (1997). A sentinel approach to fault handling in multi-agent systems. In Australian Workshop on Distributed Artificial Intelligence (pp. 181–195). Cairns, Australia: Springer Berlin Heidelberg.

Houhamdi, Z. (2011). Multi-Agent System Testing: A Survey. International Journal of Advanced Computer Science and Applications, 2(6), 135–141.

Houhamdi, Z., & Athamena, B. (2011a). Structured Integration Test Suite Generation Process for Multi-Agent System. Journal of Computer Science, 7(5), 690–697.

Houhamdi, Z., & Athamena, B. (2011b). Structured System Test Suite Generation Process for Multi-Agent System. International Journal on Computer Science and Engineering, 3(4), 1681–1688.

Houhamdi, Z., & Athamena, B. (2012). Monitoring and Diagnosis of Multi-Agent Plan : Centralized Approach. European Journal of Scientific Research, 87(4), 541–551.

Issarny, V. (2001). Concurrent Exception Handling. In Advances in Exception Handling Techniques (Vol. 2022, pp. 111–127).

Klein, M., Rodriguez-Aguilar, J. A., & Dellarocas, C. (2003). Using Domain-Independent Exception Handling Services to Enable Robust Open Multi-Agent Systems: The Case of Agent Death. Autonomous Agents and Multi-Agent Systems, 7(1/2), 179–189.

Mallya, A. U., & Singh, M. P. (2005). Modeling exceptions via commitment protocols. In Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems - AAMAS '05 (p. 122). New York, New York, USA: ACM Press.

Miller, R., & Tripathi, A. (2004). The guardian model and primitives for exception handling in distributed systems. IEEE Transactions on Software Engineering, 30(12), 1008–1022.

Romanovsky, A. (2001). Exception handling in component-based system development. In 25th Annual International Computer Software and Applications Conference. COMPSAC 2001 (pp. 580–586).

Souchon, F. F., Dony, C., Urtado, C., & Vauttier, S. (2004). Improving Exception Handling in Multi-agent Systems. In Lecture Notes in Computer Science (pp. 167–188). Springer Berlin Heidelberg.

AUTHORS BIOGRAPHY

Zina Houhamdi is an Associate Professor at the Department of Software Engineering, Al Ain University of Science and Technology, UAE. Her research work has been published in several academic journals and has been presented to scientific conferences. Her research areas of interest are data quality, agent-oriented software engineering, software testing, goal-oriented

methodology, software modelling and analysis, Petri nets, and formal methods.

Belkacem Athamena is an Associate Professor at the Department of Management and MIS, Al Ain University of Science and Technology, UAE. His research focuses on information system, data quality, data integration, software testing, software modelling and analysis, Petri nets, and formal methods. He has published in various recognized international journals and conference proceedings.